Sistema de pruebas de servicios REST mediante análisis de esquemas inferidos

Alvaro Navas, Pedro Capelastegui, Francisco Huertas, Pablo Alonso-Rodriguez, Juan Carlos Dueñas
Center por Open Middleware
Universidad Politécnica de Madrid
Campus de Montegancedo, E-28223 Pozuelo de Alarcón, Madrid, España
{alvaro.navas, pedro.capelastegui, francisco.huertas, pablo.alonso, juancarlos.duenas}@centeropenmiddleware.com

***Resumen*—**
***Palabras Clave*—jitel, telemática**

I. INTRODUCCIÓN

A lo largo de los últimos años, el paradigma de la arquitectura orientada a servicios ha tenido una gran expansión gracias a la expansión de las tecnologías web e internet. Las ventajas de esta arquitectura se basan en ofrecer diseños modulares con poco acoplamiento entre sí, lo que permite la creación eficiente y sistemática de sistemas distribuidos.

Para que este tipo de arquitectura sea posible, es necesario dotar a los servicios de interfaces de interconexión que permitan encapsular los servicios al mismo tiempo que faciliten el uso de los mismos. Existen varias tecnologías para definir estos interfaces. Entre ellas, los servicios REST, o REpresentional State Transfer, están logrando cada vez más aceptación. Esto se debe principalmente a su capacidad de escalabilidad y la uniformidad de sus interfaces, que permite una mayor separación entre los consumidores y los servicios. De hecho, compañías como Yahoo, Google o Twitter definen interfaces REST de acceso a sus servicios, ya se para consultar mapas (GoogleMaps), imágenes (Flickr) o el correo, permitiendo a terceros desarrollar clientes para sus servicios sin tener que involucrarse en su producción.

Sin embargo, los servicios REST poseen ciertas limitaciones que en ocasiones pueden complicar el desarrollo tanto del propio servicio como de clientes para el mismo. Entre estas limitaciones, destacan la falta de transparencia de la estructura y del código de los servicios, la incertidumbre sobre los componentes invocados durante la ejecución de un servicio, la falta de control sobre la infraestructura y los costes de invocación de servicio. Existen mecanismos para definir formalmente el interfaz del servicio para intentan aliviar algunas de estas limitaciones. Lamentablemente, o bien no son utilizados, o se trata de desarrollos ajenos al código del servicio y mantenidos de forma paralela, por lo que en ocasiones se pierde la sincronización entre el servicio y su descripción.

Esta falta de transparencia se traduce en que el proceso de creación de un cliente se convierta en un proceso manual que a veces incluso requiere cierto conocimiento interno del funcionamiento del servicio, que no siempre está disponible. Además de dificultar el acceso de futuros clientes a nuestro servicio, esto también dificulta la realización de pruebas sobre el mismo. Los sistemas actuales de prueba para servicios REST centran sus esfuerzos en la creación por parte del usuario de casos de prueba, tanto la llamada al servicio como la respuesta esperada, que permitan verificar el correcto funcionamiento del servicio. Dada la falta de transparencia inherente a los servicios REST, esto suele obligar a que sea el propio desarrollador del servicio el que realice las pruebas. Además, debido a que es el desarrollador el que decide qué se debe probar y qué es correcto, no se realizan pruebas sistemáticas, sino que sólo se escogen los casos que él opina que son susceptibles de producir error.

En este artículo proponemos una solución para la realización automática de pruebas de caja negra a servicios REST, lo que no sólo facilitaría la creación de casos de prueba, sino incluso la creación de clientes para servicios REST de forma automática y sin necesidad de conocer los mecanismos internos del servicio. Esta solución está basada en el análisis estadístico de las respuestas ofrecidas por el servicio al ser invocado. Este análisis se realiza en dos niveles: sintáctico y semántico. Es decir, el análisis de la estructura de la respuesta y del contenido de la misma.

Aunque el análisis estadístico de los contenidos de la respuesta podría haberse realizado sobre cualquier formato multimedia, hemos decidido centrar nuestras investigaciones en el estudio de servicios que proporcionen respuestas en formato XML. Las ventajas de esta decisión son que XML, además de ser un formato muy extendido, impone una estructura estricta y que existe abundante literatura sobre como inferirla a partir de muestras de documentos, facilitando enormemente el análisis sintáctico de las respuestas. Ambas ventajas no son necesariamente ciertas para el resto de formatos soportados por REST, por lo que XML es el candidato ideal.

El artículo sigue la siguiente estructura. La sección II ofrece una visión completa de las tecnologías usadas para realizar pruebas a servicios REST así como una visión de los diferentes tipos de inferidores de esquemas XSD existentes. La sección III muestra la arquitectura de la solución propuesta. En la sección IV describimos una serie de experimentos llevados a cabo para demostrar la validez de nuestra solución. Finalmente, la sección V contiene las conclusiones extraídas del desarrollo de este trabajo y las vías de investigación que se nos plantean de cara al futuro.

II. TRABAJOS RELACIONADOS

En esta sección vamos a revisar el estado actual de los sistemas y herramientas que permiten desarrollar pruebas para servicios REST. Así mismo, y ya que el núcleo de nuestra propuesta está basado en la inferencia de la estructura de los archivos XML, hemos considerado oportuno revisar también en qué punto de evolución se encuentran estas tecnologías.

A. Pruebas de servicios REST

Según un estudio reciente [1] , las pruebas de sistemas orientados a servicios son un tema de investigación que ha experimentado un fuerte crecimiento en los últimos años. Sin embargo, el trabajo en este área se ha centrado principalmente en servicios web basados en SOAP, mientras que las pruebas de servicios REST cuentan con un número relativamente escaso de publicaciones.

SoapUI [2] es una herramienta multiplataforma en código abierto para realizar pruebas de servicios web y servicios REST. Soporta pruebas funcionales, de regresión y de carga, e incluye funciones para pruebas de seguridad y simulación de servicios. Además, permite la generación automática de documentos WADL para la descripción de servicios REST, y la inferencia de esquemas XSD partir de las respuestas de un servicio.

TTR [3] es una herramienta de pruebas para servicios REST con soporte para pruebas funcionales y no funcionales. Se compone de un entorno extensible mediante plug-ins, un lenguaje de especificación para casos de prueba basado en XML, y un método para la composición de casos de prueba. TTR sigue un enfoque de caja negra para las pruebas, y requiere que los servicios a probar se encuentren en entornos controlados.

REST Assured [4] es un lenguaje específico de dominio para Java, desarrollado para simplificar las pruebas de servicios basados en REST. Facilita la construcción de peticiones REST, y permite validar y verificar las respuestas, con herramientas para interpretar documentos JSON y XML, soporte para autenticación, generación de trazas y mapeo de objetos.

[5] describe un entorno para pruebas automatizadas de coreografías de servicios web, que tambien es compatible con servicios REST. Se compone de un mecanismo para la abstracción de coreografías en objetos Java, clientes dinámicos para servicios web (incluyendo un cliente REST basado en REST Assured), y soporte para la intercepción de ensajes y servicios simulados.

[6] presenta un entorno software para la simulación de servicios REST en escenarios de pruebas. Para cada servicio simulado, permite la especificación de interfaces, y parámetros de entrada en forma de documentos XML. Los servicios verifican los parámetros de entrada, y generan respuestas a partir de una combinación de datos pre-grabados, lógicas definidas para el caso de prueba, y perturbación de datos.

En general, las soluciones existentes para pruebas REST presuponen un conocimiento detallado de la especificación del servicio a probar: parámetros de entrada, formato de la respuesta y valores de respuesta esperados. La necesidad de tener este conocimiento presenta varias desventajas ya comentadas en la sección anterior. Dado que la motivación del presente trabajo nace de la necesidad, en algunas ocasiones, de llevar a cabo pruebas de caja negra sobre servicios para los que se carece de esta información y de reducir el esfuerzo requerido para realizarlas, podemos concluir que ninguno de los sistemas actuales cubre estas necesidades.

B. Inferidor de esquemas XSD

Como ya hemos explicado, el análisis sintáctico de las respuestas producidas al invocar el servicio nos obliga a inferir la estructura de las mismas o, en el caso del formato XML, de su esquema. El problema de la inferencia de esquemas a partir de documentos XML se ha tratado en varios estudios a lo largo de los últimos años. Los trabajos iniciales en este área se centraban en la inferencia de esquemas de tipo DTD [7], pero en la actualidad el formato predominante es el XSD, más expresivo, y complejo.

Comparado con DTD, el lenguaje de XSD introduce propiedades como la definición de tipos o el uso de espacios de nombres que le dan mayor riqueza, pero también complican notablemente el proceso de inferencia. A grandes rasgos, mientras que en DTD el modelo de contenido de cada elemento depende únicamente de su nombre, en XSD puede depender también del contexto de uso de dicho elemento. Así, el proceso de inferencia, que en DTD se reduce a la obtención de expresiones regulares a partir de conjuntos de cadenas de elementos, requiere con XSD identificar en qué contextos se asocian diferentes modelos de contenido a cada elemento, a partir de un corpus de documentos [8]. En esta sección, resumimos los principales métodos de inferencia que han servido de referencia para el presente trabajo, y los requisitos que debería cumplir un motor de inferencia de XSD, y estudiamos las herramientas de inferencia XML más destacables.

[8] observa que los sistemas de inferencia XSD que existían previamente no tienen en cuenta el contexto de los elementos XML, lo que resulta en esquemas con sintaxis XSD, pero estructuralmente equivalentes a DTDs. Define el concepto de k-localidad, para referirse a modelos de contenido que dependen de hasta k ancestros de un elemento: así, los esquemas DTD tienen k=0, y el 99% de los XSD encontrados en la práctica tienen k=2. Además, y para simplificar el problema de inferencia, introduce una restricción adicional: usar únicamente expresiones regulares en las que cada nombre de elemento aparece una única vez, ya que el 99% de XSDs encontrados en la práctica se componen únicamente de estas expresiones regulares, a las que denomina SOREs (Single Occurrence Regular Expressions). Finalmente, propone un algoritmo de inferencia llamado iXSD, basado en SOREs y válido para cualquier k-localidad. iXSD usa autómatas de estados finitos (llamados SOA, Single Occurrence Automatons) como paso intermedio entre las cadenas de caracteres del XML y las SORE, y convierte estas últimas en el esquema deseado. Además, integra una etapa de postprocesado para unificar tipos similares en el XSD obtenido.

[9] utiliza un subconjunto de SORE más restrictivo denominado CHARE (Chain Regular Expression), que abarca la mayoría de expresiones usadas en esquemas en la práctica, e introduce un algoritmo llamado REWRITE para la transformación de autómatas en SOREs, y otro algoritmo llamado CRX, que permite inferir directamente CHAREs sin recurrir a autómatas. CRX es especialmente apropiado para escenarios con conjuntos muy reducidos de datos XML disponibles. Continuando en la misma línea, [10] propone mejoras a los algoritmos REWRITE y CRX, y un nuevo mecanismo de mezclado de tipos, que permiten inferir esquemas con el elemento sintáctico <all>, y con tipos asociados a múltiples nombre de etiqueta, respectivamente. Estos algoritmos extendidos están implementados en la herramienta Schema learner [11].

La característica principal que debemos buscar en un sistema de inferencia de esquemas XML es la capacidad de obtener esquemas válidos, precisos y compactos para tantos tipos de documento XML como sea posible, identificando correctamente los elementos, atributos y tipos del XSD, sin introducir duplicados. Para ello, el sistema de inferencia debe soportar esquemas con k-localidad de uno o mayor (para tener en cuenta el contexto de los elementos), y tener en cuenta la multitud de opciones que permite el lenguaje XSD: tipos asociados a múltiples nombres de etiqueta, identificación de elementos y atributos obligatorios, espacios de nombre, y localidad en la declaración de tipos y elementos, etc. Además, es deseable permitir inferir esquemas a partir de conjuntos numerosos de documentos XML, puesto que la cantidad de datos de entrada influye favorablemente en la calidad del esquema generado. También es importante el rendimiento, para mantener tiempos de inferencia razonablemente bajos incluso al procesar documentos XML muy abundantes y complejos.

En la actualidad, existen numerosas herramientas capaces de inferir esquemas a partir de XMLs, con diferentes niveles de prestaciones y madurez. Trang [12] es una herramienta Java con licencia New BSD capaz de inferir esquemas de distintos formatos, incluyendo DTD y XSD, así como de convertir esquemas de un tipo a otro. Como puntos negativos, sólo considera una k-localidad de cero (es decir, sus XSD son equivalentes a DTD), y los esquemas generados se desvían en ocasiones de la especificación de XSD.

El entorno .NET de Microsoft incluye desde su versión 2.0 una API de manejo de XMLs, con una clase, XmlSchemaInference [13] para la inferencia de esquemas XSD. Además, el kit de desarrollo de .NET incluye una utilidad de inferencia construida sobre esta API, llamada xsd.exe. La API toma como entrada un único documento XML y, opcionalmente, un esquema preexistente en el que integrar dicho documento, lo que permite procesar incrementalmente grupos de documentos. Sin embargo, esta aproximación supone una pérdida de información frente al procesado simultáneo de múltiples documentos, lo que resulta ocasionalmente en resultados anómalos. En nuestras pruebas de esta API, hemos observado errores moderadamente graves en ciertos escenarios, como casos en los que un ancestro comparte nombre de etiqueta con un elemento.

SoapUI [2] es una herramienta de pruebas de servicios web que, entre otras funciones, permite la inferencia de esquemas XSD a partir de las respuestas de un servicio REST. Lleva a cabo inferencias con una k-localidad de uno, y es capaz de gestionar correctamente los espacios de nombres del esquema. Al igual que .NET, permite inferir incrementalmente a partir de grupos de documentos, lo que en ocasiones produce errores. También hemos identificado algunos fallos de inferencia, por ejemplo en casos con elementos desordenados. SoapUI es el unico ejemplo previo que hemos encontrado de inferencia de XML aplicada a pruebas REST.

XML Schema Learner, cuya base teórica ya se mencionó más arriba, es una herramienta PHP en código abierto con capacidad para inferir esquemas DTD y XSD, basada en los algoritmos iXSD y REWRITE. De las herramientas estudiadas, es la que produce los esquemas más compactos y específicos.

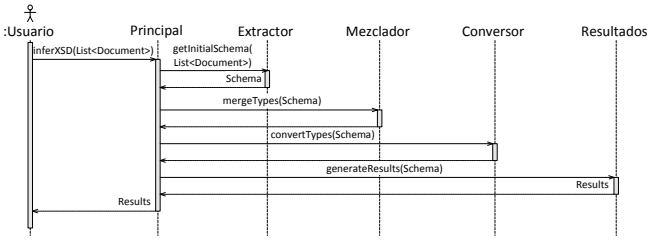

Fig. 1. Diagrama de secuencia de una inferencia de un esquema XSD

Sin embargo, la implementación ofrecida difiere enormemente de la descrita en la literatura, pudiéndose considerar incluso incompleta.

En general, las herramientas observadas sufren o bien de capacidad de inferencia limitada, o bien de errores de inferencia notables en determinados escenarios. Por esto, unido a la dificultad de integrar un módulo de obtención de estadísticas en los sistemas existentes, hemos optado por desarrollar un sistema de inferencia propio, que describimos en los siguientes apartados.

III. ARQUITECTURA DEL SISTEMA

En esta sección describiremos la arquitectura de la solución propuesta, empezando por una breve descripción del elemento principal, el inferidor de esquemas XSD, para pasar a describir la arquitectura global del sistema.

A. El inferidor de esquemas XSD

El inferidor de esquemas XSD es el corazón del sistema de detección de anomalías. Este componente es el encargado de reconstruir un esquema XSD contra el que todo el conjunto de documentos sea capaz de validar, así como de generar un conjunto de estadísticas que más tarde nos permitirán detectar anomalías tanto en la sintaxis de los archivos como en la semántica. Es por ello por lo que antes de explicar la arquitectura global del sistema, debemos describir primero cómo funciona este módulo, sin cuyo crucial trabajo no se entenderían el resto de componentes.

Como ya hemos mencionado anteriormente, después de estudiar en profundidad las diferentes aproximaciones relativas a la inferencia de esquemas, llegamos a la conclusión de que hay pocas herramientas que aprovechen la capacidad descriptiva de XSD para inferir esquemas a partir de un conjunto de documentos. Entre las que lo hacen, la herramienta más completa es la diseñada por Kore Nordmann, XML Schema Learner. Es por ello que el diseño de nuestro inferidor está fuertemente basado en el diseño de su herramienta. Sin embargo, y dado que hemos añadido el soporte para la extracción de estadísticas y modificado algunos de sus algoritmos para introducir algunas mejoras, conviene revisar la arquitectura de la herramienta.

El inferidor está diseñado como una herramienta modular independiente del resto del sistema. La Figura 1 muestra un diagrama de secuencia en el que se muestra la interacción entre los diferentes módulos y los valores que devuelve cada uno cuando se realiza una inferencia. El módulo Principal es el componente coordinador del inferidor. Su función es la de controlar el flujo de datos entre los diferentes módulo. El diseño de Kore Nordmann estaba orientado a tener una

serie de fases concretas: un extractor, un mezclador y un conversor de tipos y el módulo que crea el esquema XSD. Tras analizarlo detenidamente, nos dimos cuenta de que se podrían usar diferentes aproximaciones para resolver cada fase. Es por ello que el módulo Coordinador presenta cuatro interfaces a los que se pueden conectar diferentes módulos que pueden proveer diferentes implementaciones de los mismos. Además, también contiene la definición de las clases del modelo que se irán manipulando durante todo el proceso, como la clase Schema que contendrá la estructura de datos necesaria para inferir el esquema XSD y que se irá modificando en cada fase.

La primera fase es la extracción de tipos que realiza el módulo Extractor. El objetivo de este módulo es transformar el conjunto de documentos XML de entrada a la herramienta en una estructura de objetos que simule la estructura de tipos XSD tanto simples como complejos que debe ser capaz de contener todos los documentos. Nuestra implementación procesa los archivos XML uno a uno. Tras analizar el primer archivo, se crea un objeto Schema que contiene la estructura de elementos detectada, que incluye todos los tipos, simples y complejos, listas de atributos y su pertenencia a un determinado espacio de nombres.

El tratamiento de la mayoría de los tipos es relativamente sencilla, salvo en el caso de los tipos complejos. Estos tipos deben contener la lista de atributos correspondiente a ese tipo, el tipo simple del texto que contengan, de existir, y una representación de la estructura de sus hijos. Debido a que la estructura de estos tipos complejos no tiene por qué ser estática, por ejemplo puede haber elementos o atributos que existan en unos documentos pero no en otros, esta estructura se refleja utilizando autómatas. Con el uso de estos autómatas podemos reflejar en una estructura de objetos Java qué hijos pueden o deben aparecer en el documento y en qué orden. Además, los tipos complejos definen también qué atributos están asociados a cada elemento, incluyendo el tipo simple del atributo y si su aparición es obligatoria o no.

La estructura de elementos almacenada en el objeto Schema hace uso de estos tipos complejos a la hora de definir la secuencia de elementos. Los archivos sucesivos se utilizan para refinar los elementos ya presentes o crear nuevos en caso de ser necesario y para modificar la estructura del documento. El resultado final es un objeto Schema que contiene una versión primitiva del esquema XSD final que contiene una lista de todos los tipos simples y complejos detectados y la lista de los diferentes espacios de nombre encontrados, con todos los prefijos con los que han aparecido.

Una mejora incluida en esta implementación frente a la de Kore Nordmann es que somos capaces de detectar un número mayor de tipos simples que su herramienta, que clasificaba todos los tipos como xs:string. Además, podemos también determinar cuando un tipo XSD tiene una restricción asociada de tipo Enumeration. Además, también hemos introducido una modificación que nos permite trabajar con diferentes espacios de nombre. Finalmente, debemos mencionar que en cada análisis se extrae información relativa a la aparición de cada elemento y los diferentes valores que puede tomar, que se almacena en una estructura de objetos Java auxiliar dentro del objeto Schema y que nos servirá para elaborar un informe con las estadísticas extraídas del cuerpo de documentos al final del proceso.

El siguiente paso es la mezcla de tipos que realiza el módulo Mezclador. La salida del Extractor es un conjunto de objetos Java interrelacionados que conforman un esquema XSD primitivo. El objetivo de este módulo es analizar este esquema y unificar aquellos tipos que son lo suficientemente parecidos como para ser considerados un único tipo XSD, reduciendo enormemente el tamaño del esquema. Este proceso es bastante complejo ya que requiere la comparación de todos los tipos. Para los tipos simples eso significa comparar todas las listas de valores. Para los complejos, hay que comparar las estructuras de sus autómatas y las listas de atributos. Además, el proceso debe ser especialmente cuidadoso para poder preservar los datos estadísticos extraídos durante la fase anterior y que no se pierdan durante la mezcla.

El siguiente paso para transformar la estructura de objetos Java en un esquema XSD es convertir nuestra estructura de tipos en un una serie de expresiones regulares a partir de las cuales se puede construir el esquema XSD definitivo. Esta tarea corre a cargo del módulo Conversor. Este módulo se compone de dos partes diferenciadas. La primera está destinada a crear las expresiones regulares. Para ello se pueden utilizar varios métodos de extracción, principalmente basados en la extracción de SOREs o CHAREs. Ambos tipos de expresiones regulares garantizan una estructura no ambigua y sin dependencias cíclicas, usándose las SOREs por defecto, ya que son más restrictivas, y las CHAREs en caso de que el proceso de creación de la SORE no se pueda completar o cuando se defina así en la configuración de la herramienta. De esta forma nos aseguramos de que el esquema está lo más restringido posible a la estructura de los documentos analizados, evitando dar como válidos documentos que pudieran tener una estructura parecida, pero que no fuera válida.

Las expresiones regulares resultado de extraer utilizando cualquiera de los dos métodos suelen contener información redundante o innecesaria, por lo que la segunda parte de este módulo está dedicada a optimizar las expresiones regulares resultantes hasta que contengan sólo la información mínima necesaria. En concreto, se utilizan optimizadores para eliminar elementos vacíos, para reducir la nomenclatura producida (en especial la concatenación de dos multiplicadores) e incluso para reducir la longitud de algunas secuencias. En la figura 2 se puede observar un ejemplo de autómata convertido a expresión regular siguiendo este método.

Finalmente, el módulo Resultados se encarga de generar los documentos de salida de la herramienta. En concreto, genera un esquema XSD por cada namespace diferente que se haya encontrado dentro del cuerpo de documentos así como un informe estadístico separado en el que se analizan diferentes datos sobre cada elemento. Por cada elemento único encontrado en todo el cuerpo de documentos se calcula su número total de apariciones diferentes, así como su media, máxima y mínima por archivo. Sobre estos datos se puede calcular además la moda y la varianza dependiendo de si el tipo de datos de ese nodo lo admite. Por ejemplo, si es un nodo de tipo XSD Integer, podemos averiguar cual es el valor más común a partir de estos datos y cual es el rango de variación a partir de la desviación.

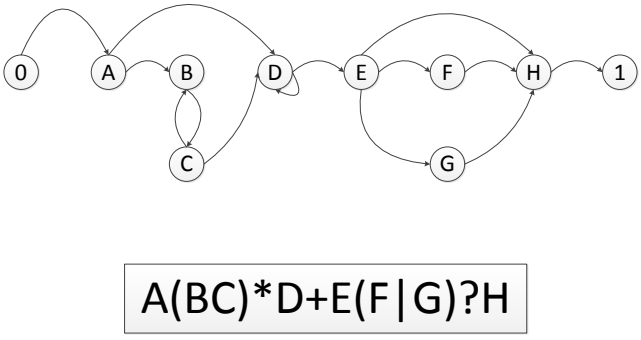

Fig. 2. Conversión de un autómata a expresión regular

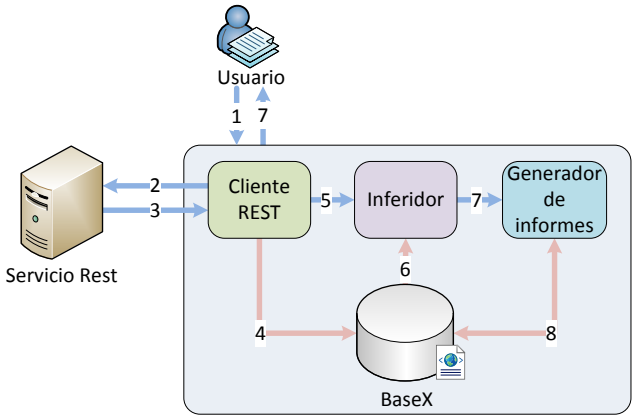

Fig. 3. Arquitectura global de la herramienta

B. *Arquitectura global*

Una vez explicada la pieza central del sistema y cómo genera las estadísticas, podemos pasar a explicar la arquitectura global del mismo. Recordemos que el objetivo de la herramienta es la de poder realizar pruebas de caja negra sobre servicios Rest, basada en la detección estadística de elementos anómalos. Es por ello que el sistema debe componerse de tres módulos: uno capaz de realizar llamadas a un servicio REST cualquiera y procesar las respuestas; el inferidor, que analiza las respuestas y calcula las estadísticas de cada elemento; y un módulo final que analice las estadísticas extraídas para detectar aquellos elementos, tanto sintácticos como semánticos, que destaquen respecto al resto. La Figura 3 presenta la visión global del sistema así como la secuencia de acciones que se llevan a cabo cuando se quiere ejecutar un análisis de un servicio REST.

El primer paso es la introducción por parte del usuario de un conjunto de datos de prueba. Este conjunto de datos de prueba consiste simplemente en la URL y el método HTTP del servicio que se quiere probar así como un conjunto de parámetros que permitan variar la petición. Cuanto mayor sea este conjunto de parámetros, mejor serán tanto la detección de anomalías como el XSD inferido.

Los datos de entrada llegan al módulo Cliente REST. Este módulo lee estos datos y genera peticiones REST al servicio especificado. Por cada respuesta que recibe, crea un documento XML que asocia esa respuesta a los parámetros utilizados que generaron esa respuesta y almacena ambos en la base de datos del sistema. Para el almacenamiento temporal de los datos hemos utilizado una base de datos orientada a documentos XML llamada BaseX [14], dado que queremos poder realizar búsquedas sobre el cuerpo de documentos sin tener que adaptarlos a un esquema SQL.

Cuando se han finalizado todas las peticiones y todos los resultados se encuentran almacenados en la base de datos, el Cliente REST procede a decirle al Inferidor en que parte de la base de datos puede encontrar el cuerpo de documentos que debe analizar. El funcionamiento del Inferidor ya fue detallado en la sección anterior, por lo que no volveremos a explicar su funcionamiento. Sólo añadiremos que recoge el conjunto de documentos de entrada de la base de datos y almacena los resultados en la misma.

Finalmente, el módulo Generador de Informes procede a analizar los resultados estadísticos recogidos por el Inferidor para detectar anomalías. Para ello se sirve de un conjunto de reglas que aplica a nivel sintáctico y semántico. A nivel sintáctico recorre el esquema XSD para detectar aquellos tipos que tienen un número de apariciones mucho menor que el del resto de elementos. Por ejemplo, si tras analizar 20 documentos encontrásemos un tipo que sólo ha aparecido dos veces, se podría tratar de un error, especialmente si se detecta que es un tipo que existe en otra parte del esquema, lo que indicaría un error humano al crear el documento. A nivel semántico, el descubrimiento de anomalías sigue reglas algo más complejas en las que se comparan diversas estadísticas de los valores que toma cada elemento, tales como la varianza, la moda, el máximo y el mínimo número de apariciones por archivo... Una vez tenemos esos valores, se procede a comparar los mismos entre sí, como en el caso de la varianza y los máximos y mínimos para saber si existe duplicidad de algún valor, o con los de sus valores hermanos, por ejemplo cuando en la lista de valores de un elemento tiene un valor muy distinto al de sus hermanos, ya sea superior o inferior.

Tras descubrir las posibles anomalías, se escribe un informe en el que se recogen todas las anomalías detectadas y su relación con los parámetros de entrada que los produjeron. Para ello se hace uso de las funcionalidades de búsqueda que ofrece BaseX basados en XQuery, un lenguaje de consultas específico para archivos XML. El módulo recoge la ruta XML que generó la anomalía y realiza una búsqueda sobre el conjunto de documentos almacenados en la base de datos. Con los archivos que generaron la anomalía identificados, procede a leer la información introducida por el módulo Cliente REST que ataba esos archivos a parámetros de entrada para reflejar dicha relación en el informe final. Cuando todas las anomalías han sido procesadas, devuelve al usuario tanto el informe como los esquemas XSD que se hayan generado. El motivo de devolver los esquemas XSD no es meramente informativo, ya que al usuario podría usarlos para crear un cliente REST capaz de validar e interpretar todas las respuestas del servidor a partir de los mismos si así lo desea.

IV. VALIDACIÓN

En esta sección describiremos las pruebas realizadas para comprobar la validez de la solución propuesta. Para ello, se han realizado dos tipos de pruebas. Las primeras se han cen-

Tabla I
CARACTERÍSTICAS DE LOS SERVICIOS REST UTILIZADOS

Servicio	Google Places	Servicio REST propio
Proveedor	Google	Propio
Funcionalidad	Encontrar puntos de interés próximos a una posición facilitada	Obtener información de configuración de un servidor web
Tipo de respuesta	corta y homogénea	larga y homogénea
Estado del servicio	Muy estable	En fase de pruebas

Tabla II
PARAMETROS PARA EL SERVICIO REST GOOGLE PLACES

Nombre	Valor
Location	Localización aleatoria dentro de la península ibérica
Radius	Radio entre los parámetros permitidos por el servicio (10-50000 m.)
Languaje	Idioma de respuesta aleatorio entre los admitidos por el servicio.
Type	tipo de local buscado aleatorio entre el conjunto tipos soportados.
Query	spain
Sensor	false

trado en validar el funcionamiento del inferidor de esquemas, ya que al ser la pieza central del sistema debemos asegurar su correcto funcionamiento. Estas pruebas han sido realizadas tanto sobre nuestros sistema como sobre la herramienta de código libre SoapUI, que como ya hemos mencionado permite la inferencia de esquemas a partir de las respuestas de un servicio REST. Las pruebas pertenecientes al segundo tipo se han centrado en probar la validez del sistema completo, estudiando las anomalías detectadas en servicios REST reales. Estas pruebas se han realizado sólo sobre nuestra herramienta al ser una funcionalidad que no posee ningún otro sistema.

A. Escenario

Para la realización de las pruebas se han definido dos escenarios diferentes. Para el primero hemos seleccionado un servicio REST público que nos permitirá comprobar cómo se comporta la herramienta cuando el usuario desconoce el funcionamiento interno del servicio. Sin embargo, dado que se trata de un servicio público en funcionamiento, no esperamos poder encontrar errores en su funcionamiento, por lo que para el segundo escenario hemos decidido utilizar un servicio REST propio que se encuentra actualmente en desarrollo y puede contener errores. En ambos casos, se han realizado 50 peticiones diferentes a cada servicio.

1) Google Places: Como primer servicio REST se ha tomado el servicio público ofrecido por Google, Google Places [15], cuyas características vienen indicadas en la tabla I.

En este escenario se han realizado varias consultas al servicio para intentar encontrar diferentes tipos de locales en situaciones aleatorias a lo largo de la península ibérica. Los parámetros de entrada al servicio están indicados en la tabla II.

2) Servicio REST propio: Para el segundo escenario hemos elegido un servicio REST desarrollado por nosotros que se encuentra ahora mismo en fase de pruebas. El servicio permite consultar la configuración de diversos servidores de aplicación distribuidos en nuestra nube privada formada por seis máquinas físicas con capacidad para contener hasta 10 máquinas virtuales cada una. Las características del servicio se pueden consultar en la última columna de la tabla **??**. El único parámetro de entrada requerido por el servicio es la dirección del servidor cuya configuración se desea consultar.

B. Validación de la inferencia

Los objetivos fijados para estas pruebas son conocer el número mínimo de respuestas necesarias para generar un esquema y comparar los esquemas obtenidos por las dos herramientas.

1) :: Google Places

Nuestra herramienta obtiene un esquema que valida la mayor parte de las respuestas con el conjunto inicial de respuestas. Las respuestas que no validan el esquema son respuestas poco usuales, concretamente respuestas sin resultados y, al incluir una de estas respuestas al conjunto de respuestas, el esquema generado valida todas las respuestas. SoapUI es capaz de generar un esquema que valida la mayor parte de las respuestas con una única respuesta, es necesario añadir información de 5 respuestas adicionales que añaden información referente al número de apariciones posibles de elementos. Nuestra herramienta necesita por tanto más respuestas que SoapUI para crear un esquema valido, debido principalmente a la capacidad de identificar enumeraciones, sin embargo esta necesidad esta cubierta en gran parte por el grupo inicial y, en este caso, solo es necesario incluir la información de una respuesta adicional. SoapUI sin embargo, necesita un grupo de respuestas menor para crear un esquema que valide todas las respuestas, 11 contra 6, necesita realizar más iteraciones, una por cada respuesta adicional, para obtener el esquema valido.

Si comparamos los esquemas XSD resultantes podemos observar que son muy similares, nuestra herramienta tiene información adicional de enumeraciones, en concreto ha identificado un elemento que puede puede tener dos valores que indican si la búsqueda ha obtenido resultados o no.

2) :: Servicio REST propio

Nuestra herramienta obtiene, a partir de el grupo inicial de respuestas, un esquema XSD que valida la mayoría de las respuestas, en concreto no valida dos de las respuestas y la causa de que no validen es que poseen elementos que descolocados en la estructura XML. SoapUI no es capaz de generar ningún esquema a partir de las respuestas obtenidas debido a que es incapaz de inferir esquemas cuando se encuentran elementos hermanos del mismo tipo intercalados.

En este servicio SoapUI es incapaz de generar un esquema y, por lo tanto, no se pueden comparar los esquemas. Aún así hay que señalar que nuestra herramienta ha sido capaz de extraer 16 enumeraciones distintas de la respuesta del servicio.

C. Análisis de anomalías

En estas pruebas hemos decidido comprobar la funcionalidad principal del sistema, la obtención de anomalías a partir de información estadística extraída de las respuestas del servicio. Para validar esta funcionalidad se han obtenido de la herramienta los informes de anomalías correspondientes a analizar las 50 peticiones. Posteriormente, se ha analizado

Fig. 4. Extracto de las estadísticas obtenidas de las respuestas de Google Places.

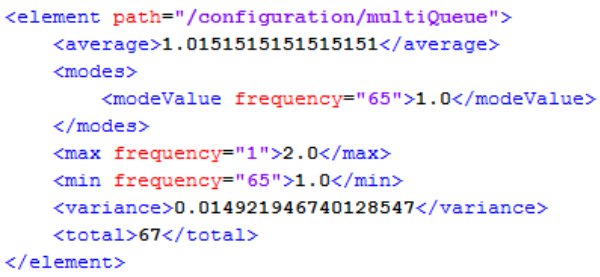

Fig. 5. Elemento repetido.

cada anomalía para determinar su origen y si se trata de un fallo del servicio o no.

1) Google places: En el análisis de las respuestas del servicio REST se ha observado que existe una anomalía. Esta anomalía, figura 4, consiste en un índice de aparición bajo de un valor en un campo en donde existen pocos valores, en concreto una aparición de 5 ocurrencias sobre 50 con 2 posibles valores para este campo de la respuesta en el servicio REST. Un análisis de las respuestas que generan la anomalía revela que esta se trata de un falso positivo causado por una respuesta poco común, pero válida, del servicio.

2) Servicio REST propio: En este servicio REST se esperaba que la herramienta obtenga anomalías a partir de las respuestas obtenidas. Como resultado de análisis de la herramienta se han obtenido tres tipos de anomalías.

En el primer tipo de anomalía, figura 5, la frecuencia más común del elemento que produce la anomalía es una

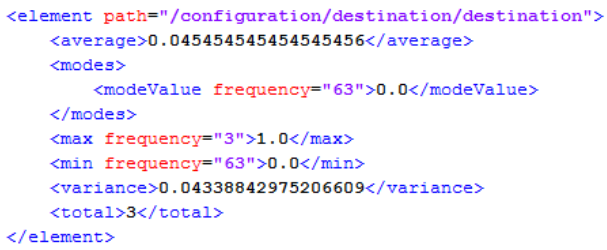

Fig. 6. Elemento incorrecto

única aparición; sin embargo, una de las respuestas tiene dos elementos. Tras estudiar la anomalía se han determinado que en todos los casos el elemento estaba duplicado y, por lo tanto, era necesario eliminarlo de la respuesta. El valor estadístico que ha identificado esta anomalía ha sido la varianza, la cual tiene un valor muy bajo, sin ser 0.

En la segunda anomalía, se puede observar como existe un elemento en casi todas las respuestas del servicio REST y la frecuencia de aparición en estos casos es siempre uno. Tras el estudio de la anomalía se ha observado que este elemento debería estar en todas las respuestas. El valor estadístico que ha identificado esta anomalía ha sido la varianza la cual tiene un valor muy bajo, sin ser 0.

En el último tipo de anomalía, figura 6, se presenta en un elemento que aparece solo en un número muy pequeño de resultados del servicio. Cuando se analiza las respuestas se ha identificado que existen varias razones que ocasionan esta anomalía. Una de ellas corresponde a un falso positivo, ya que al igual que ocurrió en el análisis del servicio REST Google Places, se trata de respuestas atípicas. Otra razón ha sido que el elemento no debería existir y, por lo tanto, se ha obtenido una respuesta incorrecta del servicio REST. La última de las razones que ha causado una anomalía ha sido un elemento mal situado dentro de la respuesta o un nombre de un elemento incorrecto; en este caso, pese a ser una repuesta incorrecta, ya ha sido detectado en el anterior tipo de anomalía y era producido por elementos que faltaban en las respuestas del servicio. Este tipo de anomalía se ha identificado gracias al conjunto de las variables estadísticas que indican el valor máximo y mínimo de apariciones de los elementos.

Como resultado del análisis de las anomalías se han detectado:

- Tres elementos duplicados en diferentes respuestas XML del servicio REST analizado.
- Cinco elementos descolocados en varias de las respuestas XML del servicio REST analizado.
- Un elemento correctamente colocado pero que el nombre se encuentra modificado en una de las respuestas XML del servicio REST analizado.
- Dos elementos que no deberían aparecer en las respuestas XML del servicio REST analizado.
- Siete casos en los que la anomalía se debía a un falso positivo.

V. CONCLUSIONES Y TRABAJOS FUTUROS

Como hemos visto en la sección anterior, nuestra solución es capaz de descubrir anomalías que otros sistemas de pruebas no son capaces de detectar con la ventaja de que no se necesita

especificar la respuesta esperada para cada petición. Además, obtenemos un esquema XSD de forma más consistente y más específica que otras herramientas de inferencia disponibles en el mercado, lo que permitirá en un futuro generar de forma automática la especificación WADL del servicio o incluso crear de forma automática las clases de un cliente específico para ese servicio, reduciendo enormemente el tiempo de desarrollo. También hemos podido comprobar que la herramienta es capaz de detectar anomalías que, si bien no son errores, pueden determinar casos especiales, ofreciendo la posibilidad de utilizar el sistema para detectar cuales son esos casos especiales. Por ejemplo, en el caso de estudio sobre Google Places, podría ser interesante encontrar esos lugares en los que no hay locales alrededor. Esto nos lleva a pensar que se podrían explorar otras funcionalidades para la herramienta.

Sin embargo, este sistema presenta algunas desventajas. Debido a que el análisis es estadístico, no se puede utilizar para realizar pruebas simples, con pocas peticiones al servicio, ni para detectar errores que no destaquen del funcionamiento normal, es decir, errores que sucedan siempre o en una proporción igual a la de otros errores. Una posible solución a este problema sería la posibilidad de incluir como parámetro de entrada el esquema XSD esperado y compararlo con el obtenido para detectar los elementos anómalos.

Para que esta solución fuese posible, habría que asegurar que el esquema inferido es lo más fiel posible a la realidad. Aunque nuestro inferidor ofrece esquemas muy definidos y atados a los documentos de entrada, se podría mejorar. Por ejemplo, el algoritmo que calcula la distancia entre elementos XML que se utiliza para calcular los elementos que pertenecen a un tipo complejo podría ser sustituida por nuevos algoritmos como el calculo de la distancia pq-gram [16]. Cambios de este tipo deberían ser estudiados para cada uno de los módulos que componen el inferidor para así mejorar en la medida de lo posible el esquema resultante del proceso.

La otra principal desventaja que presenta el sistema es que su funcionalidad se limita al estudio de las respuestas XML de un servicio REST. Otra vía de evolución consistiría en estudiar la forma de aplicar el mismo tratamiento a respuestas JSON o en otros formatos. Tanto la solución a esta desventaja como a la anterior representan vías de investigación que nos gustaría abordar en un futuro, ahora que hemos determinado la viabilidad de la solución, para así poder generalizar el uso de la herramienta.

Otra vía de investigación a considerar sería la mejora del módulo detector de anomalías. El estudio de reglas más complejas y de nuevos mecanismos de detección, como algoritmos de aprendizaje automatizado tales como algoritmos de clasificación o de detección de grupos, supondrían una mejora de la calidad de los resultados ofrecidos. Estos nuevos mecanismos permitirían detectar más errores reduciendo al mismo tiempo el número de falsos positivos. También sería interesante investigar la forma de transmitir de forma más concisa el diagnóstico de una anomalía, ya que ahora tan sólo señala los elementos anómalos, dejando al usuario la responsabilidad de averiguar por qué destaca ese elemento.

Para finalizar, pensamos que este sistema es sólo el primer paso para crear un entorno de soporte al desarrollo de servicios REST, ofreciendo la posibilidad no sólo de depurar el servicio sino además ofrecer funcionalidades adicionales como la generación de la definición del servicio o la generación de un cliente.

REFERENCIAS

[1] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, 2012. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/stvr.1470/full

[2] "SoapUI - the home of functional testing," http://www.soapui.org/. [Online]. Available: http://www.soapui.org/

[3] S. Chakrabarti and P. Kumar, "Test-the-REST: an approach to testing RESTful web-services," in *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:*, 2009, pp. 302–308.

[4] "rest-assured - java DSL for easy testing of REST services," http://code.google.com/p/rest-assured/. [Online]. Available: http://code.google.com/p/rest-assured/

[5] F. Besson, P. Leal, and F. Kon, "Rehearsal: A framework for automated testing of choreographies," Technical report, University of Sao Paulo, Department of Computer Science, Tech. Rep., 2011. [Online]. Available: http://ccsl.ime.usp.br/baile/files/tech-report-vv_2011.pdf

[6] H. Reza and D. Van Gilst, "A framework for testing RESTful web services," in *2010 Seventh International Conference on Information Technology: New Generations (ITNG)*, 2010, pp. 216–221.

[7] Y. Papakonstantinou and V. Vianu, "DTD inference for views of XML data," in *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2000, pp. 35–46.

[8] G. J. Bex, F. Neven, and S. Vansummeren, "Inferring XML schema definitions from XML data," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 998–1009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325964

[9] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of concise regular expressions and DTDs," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 2, p. 11, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1735890

[10] K. Nordmann, "Algorithmic learning of XML schema definitions from XML data," 2011. [Online]. Available: https://kore-nordmann.de/talks/11_03_learning_xml_schema_definitions_from_xml_data.pdf

[11] "XML-Schema-learner," https://github.com/kore/XML-Schema-learner. [Online]. Available: https://github.com/kore/XML-Schema-learner

[12] "Trang," http://www.thaiopensource.com/relaxng/trang.html. [Online]. Available: http://www.thaiopensource.com/relaxng/trang.html

[13] "Microsoft .NET framework," http://www.microsoft.com/net. [Online]. Available: http://www.microsoft.com/net

[14] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl, *XQuery full text implementation in BaseX*. Springer, 2009.

[15] "Api del servicio web de google places," Website, https://developers.google.com/places/documentation/l.

[16] N. Augsten, M. Böhlen, and J. Gamper, "The pq-gram distance between ordered labeled trees," *ACM Transactions on Database Systems (TODS)*, vol. 35, no. 1, p. 4, 2010.